

DOM Manipulation | CRUD

(Detailed notes yet beginner friendly)

In-Depth Guide to DOM Manipulation

Once you have selected an element and stored it in a variable, you have a powerful object that gives you full control over that part of your webpage. Let's cover the three main categories of manipulation:

1. **Editing What's Inside (Content & HTML)**
2. **Editing the Tag Itself (Attributes, Classes, and Styles)**
3. **Editing the Page Structure (Creating, Adding, and Removing Elements)**

Part 1: Editing the Content Inside an Element

This is about what the user sees inside an element's opening and closing tags.

HTML Snippet for this section:

```
<div id="welcome-box">
  Please log in to see your messages.
</div>
```

```
// We've selected our target element
const welcomeBox = document.getElementById('welcome-
box');
```

A. `.textContent` (For Plain Text - The Safe Default)

- **What it does:** Gets or sets the **pure text** content of an element. It ignores and strips out all HTML tags.
- **Use Case:** This should be your **default choice** for changing text on the page.

```
// Reading the text content
console.log(welcomeBox.textContent); // "Please log in to
see your messages."
```

```
// Writing new text content
welcomeBox.textContent = "Welcome back, Alice!";
```

The Security Benefit :When you set `.textContent` , the browser treats your string as plain text. This is safe because it prevents malicious code from being executed.

```
// Even if this string came from a hacker...
const maliciousInput = "<script>alert('hacked!');</script>";

welcomeBox.textContent = maliciousInput;
// The browser will LITERALLY display the text: "
<script>alert('hacked!');</script>"
// It will NOT run the script.
```

B. `.innerHTML` (For HTML - Powerful but Dangerous)

- **What it does:** Gets or sets the **full HTML markup** inside an element.
- **Use Case:** Use this **only when you need to generate HTML** and the content is from a **trusted source** (i.e., you, the developer).

```
// Let's create a more complex message
const welcomeMessage = `
  <h2>Welcome back, Alice!</h2>
  <p>You have <strong>5</strong> new messages.</p>
`;

// Writing new HTML content
welcomeBox.innerHTML = welcomeMessage;
```

The Security Risk (XSS - Cross-Site Scripting) : Never use `.innerHTML` with content provided by a user (like a comment or a username). If a user enters malicious `<script>` code, `.innerHTML` will execute it, allowing the user to attack your website and its visitors.

Part 2: Editing the Element's Tag Itself

This is about changing the attributes of the tag, like its id, class, or style.

HTML Snippet for this section:

```
<div id="profile-pic-container" class="card">  
    
</div>
```

```
const profileContainer = document.getElementById('profile-  
pic-container');  
const profileImage = profileContainer.querySelector('img');
```

A. Changing Common Attributes (Direct Properties)

For standard attributes, you can often change them directly as properties on the element object.

```
// Change the image source
profileImage.src = 'images/alice.png';

// Change the alt text for accessibility
profileImage.alt = 'A photo of Alice';

// Change the ID of the container
profileContainer.id = 'user-123-avatar';
```

B. Changing CSS Classes (The `.classList` Toolbox)

This is the best and safest way to manage an element's classes. Forget about **`.className`**

`.add('className')` : Adds a new class.

`.remove('className')` : Removes a class.

`.toggle('className')` : Adds the class if it's missing, removes it if it's there.

`.contains('className')` : Checks if an element has a class (returns **true** or **false**).

```
// Let's highlight the profile picture when the user is active
profileContainer.classList.add('is-active'); // Adds the 'is-active' class

// Let's remove the generic 'card' class
profileContainer.classList.remove('card');

// We can toggle a 'selected' state
profileContainer.classList.toggle('selected'); // Adds 'selected'
profileContainer.classList.toggle('selected'); // Removes 'selected'
```

Best Practice : Define your styles in a CSS file and use JavaScript's **.classList** to apply or remove those styles. This keeps your presentation (CSS) and logic (JS) separate.

C. Changing Inline Styles (The **.style** Property)

This allows you to directly apply CSS styles to an element's **style** attribute.

- **CRITICAL RULE**: CSS properties with a hyphen are converted to camelCase.
 - **background-color** becomes **backgroundColor**.
 - **font-size** becomes **fontSize**.

```
const profileContainer = document.getElementById('profile-pic-container');

profileContainer.style.border = '2px solid blue';
profileContainer.style.borderRadius = '50%'; // Note the camelCase
profileContainer.style.width = '150px';    // Values must be strings with units
profileContainer.style.height = '150px';
```

Best Practice : Best for dynamic styles that are calculated at runtime (e.g., setting an element's position based on the mouse). For static styles, prefer using **.classList**.

Part 3: Editing the Page Structure

This is the most powerful form of DOM manipulation.

A. Creating a New Element: `document.createElement()`

This creates a new element node in memory. It is **not yet on the page**.

```
// Create a new paragraph, but it's just floating in memory.
const newParagraph = document.createElement('p');

// Now, we configure it.
newParagraph.textContent = "This is a brand new paragraph created by
JavaScript.";
newParagraph.classList.add('info-text');
```


B. Adding the New Element to the Page

HTML:

```
<div id="parent-container">
  <p id="first-child">First Paragraph</p>
  <p id="reference-element">I am the reference point.</p>
  <p id="last-child">Last Paragraph</p>
</div>
```

JavaScript:

```
// This is the new element we want to add.
const newElement = document.createElement("h2");
newElement.textContent = "I AM THE NEW ELEMENT";

// This is our main reference point for the examples.
const referenceElement = document.getElementById('reference-element');
```

Here are all the different ways you can add **newElement** to the page.

1. Adding Inside a Parent (as a Child)

These methods place the new element inside the boundaries of another element.

.append(...nodes) (Modern & Recommended)

What it does : Inserts the node as the very last child of the parent.

Use Case : The default way to add an item to the end of a list or container.

```
const parent = document.getElementById('parent-container');
parent.append(newElement);
```

Resulting HTML:

```
<div id="parent-container">
  <p id="first-child">...</p>
  <p id="reference-element">...</p>
  <p id="last-child">...</p>
  <h2>I AM THE NEW ELEMENT</h2> <!-- Appended here -->
</div>
```

.prepend(...nodes) (Modern & Recommended)

What it does : Inserts the node as the **very first child** of the parent.

Use Case : Adding a new item to the top of a feed or a list.

```
const parent = document.getElementById('parent-container');
parent.prepend(newElement);
```

Resulting HTML:

```
<div id="parent-container">
  <h2>I AM THE NEW ELEMENT</h2> <!-- Prepended here -->
  <p id="first-child">...</p>
  <p id="reference-element">...</p>
  <p id="last-child">...</p>
</div>
```

.appendChild(node) (Classic)

What it does : Same as **append**, but it's the older syntax. It can only add one node at a time and returns the appended node.

```
const parent = document.getElementById('parent-container');
parent.appendChild(newElement); // Same result as append
```

.insertBefore(newNode, referenceNode) (Classic & Powerful)

What it does : Inserts newNode into the parentElement right before the referenceNode . This is the classic way to insert into the middle of a list of children.

Use Case : Inserting an element at a specific position in a list.

```
const parent = document.getElementById('parent-container');
const referenceNode = document.getElementById('reference-element');
parent.insertBefore(newElement, referenceNode);
```

Resulting HTML:

```
<div id="parent-container">
  <p id="first-child">...</p>
  <h2>I AM THE NEW ELEMENT</h2> <!-- Inserted before the reference -->
  <p id="reference-element">...</p>
  <p id="last-child">...</p>
</div>
```

2. Adding Next to an Element (as a Sibling)

These methods place the new element outside the boundaries of the reference element, at the same level in the DOM tree.

.after(...nodes) (Modern & Recommended)

What it does : Inserts the node immediately after the reference element, as its next sibling.

Use Case : Adding a new element directly following another one.

```
const referenceElement = document.getElementById('reference-element');
referenceElement.after(newElement);
```

Resulting HTML:

```
<div id="parent-container">
  <p id="first-child">...</p>
  <p id="reference-element">...</p>
  <h2>I AM THE NEW ELEMENT</h2> <!-- Inserted here, as a sibling -->
  <p id="last-child">...</p>
</div>
```

.before(...nodes) (Modern & Recommended)

What it does : Inserts the node immediately before the reference element, as its previous sibling.

Use Case : Adding a new element directly preceding another one.

```
const referenceElement = document.getElementById('reference-element');
referenceElement.before(newElement);
```

Resulting HTML:

```
<div id="parent-container">
  <p id="first-child">...</p>
  <h2>I AM THE NEW ELEMENT</h2> <!-- Inserted here, as a sibling -->
  <p id="reference-element">...</p>
  <p id="last-child">...</p>
</div>
```

.insertAdjacentElement(position, element) (The Most Versatile)

What it does : This is a highly flexible method that can do the job of all the others. You specify a position relative to the target element and the element to insert.

- **The Four Positions:**
- **'beforebegin'** : Inserts the new element as a sibling, right before the target element (same as `.before()`).
- **'afterbegin'** : Inserts the new element as the first child of the target element (same as `.prepend()`).
- **'beforeend'** : Inserts the new element as the last child of the target element (same as `.append()`).
- **'afterend'** : Inserts the new element as a sibling, right after the target element (same as `.after()`).

Example (replicating `.after()`):

```
const referenceElement = document.getElementById('reference-element');
referenceElement.insertAdjacentElement('afterend', newElement);
```

Example (replicating `.prepend()`):

```
const parent = document.getElementById('parent-container');
// Note: we are calling this on the parent to insert AS A CHILD
parent.insertAdjacentElement('afterbegin', newElement);
```

There is also a related method, `.insertAdjacentHTML()`, which works like `.innerHTML` for these specific positions, but it carries the same security risks.

Summary Table: Which Method to Use?

Goal	Best Modern Method	Best Modern Method
Add as the LAST child	parent.append(el)	parent.appendChild(el)
Add as the FIRST child	parent.prepend(el)	parent.insertBefore(el, parent.firstChild)
Add BEFORE another element	referenceEl.before(el)	parent.insertBefore(el, referenceEl)
Add AFTER another element	referenceEl.after(el)	parent.insertBefore(el, referenceEl.nextElementSibling)

C. Removing an Element

- **.remove()** : The simplest and most modern way. You call it directly on the element you want to remove.

```
// Let's say we want to remove the profile image.  
const profileImage = document.querySelector('#user-123-avatar img');  
  
// First, check if the element was actually found before trying to remove  
it.  
if (profileImage) {  
  profileImage.remove(); // The image is now gone from the page.  
}
```

This comprehensive set of tools (**.textContent**, **.classList**, **.style**, **createElement**, **append**, **remove**) gives you everything you need to transform a static HTML page into a fully dynamic application.

The First Thought:

"Every time I touch the live webpage, it's expensive. I should do all my work 'offline' on a temporary, invisible scratchpad, and then attach the finished result to the live page in one single, efficient step."

That "invisible scratchpad" is a **DocumentFragment**.

The Problem: The High Cost of Touching the "Live" DOM

The DOM that is visible on the webpage is "live." Every time you change it—by appending a child, changing a style that affects size, etc.—you force the browser to do work.

1. **DOM Update** : The element is added to the tree.
2. **Reflow/Layout** : The browser may have to recalculate the position and size of many elements on the page.
3. **Repaint** : The browser has to repaint the pixels for the updated part of the screen.

Now, imagine you need to add 1,000 items to a list.

The Inefficient "Bad" Way:

This code touches the live DOM in a loop.

```
const list = document.getElementById('my-list'); // This is a "live" DOM element.
```

```
// START THE TIMER
```

```
console.time("Loop without fragment");
```

```
for (let i = 0; i < 1000; i++) {  
  const newItem = document.createElement('li');  
  newItem.textContent = `Item ${i + 1}`;
```

```
  
  // This is the expensive part. We are modifying the live DOM 1,000 times.
```

```
  // This can cause up to 1,000 separate reflows and repaints.  
  list.appendChild(newItem);  
}
```

```
// STOP THE TIMER
```

```
console.timeEnd("Loop without fragment");
```

If you run this, you might notice a brief stutter or freeze in the browser. You are forcing the browser to do a massive amount of repetitive work. Each **appendChild** is a separate transaction with the rendering engine.

The Solution: The `DocumentFragment`

A **`DocumentFragment`** is a **lightweight, in-memory DOM node that is not part of the main document tree**. It's a temporary, invisible container.

- It has the same API as a regular element: you can **`.appendChild()`** or **`.append()`** to it.
- **The key difference** : Modifying a **`DocumentFragment`** is extremely cheap. Since it's not part of the visible page, changing it does not trigger any reflows or repaints.

The Efficient "Good" Way : This code does all the heavy lifting "offline" and then updates the live DOM only once.

```
const list = document.getElementById('my-list');

// START THE TIMER
console.time("Loop WITH fragment");

// 1. Create the invisible "scratchpad".
const fragment = document.createDocumentFragment();

// 2. Do all your work on the scratchpad.
// This loop is now very fast because it's only changing things in
memory,
// not on the visible page.
for (let i = 0; i < 1000; i++) {
  const newItem = document.createElement('li');
  newItem.textContent = `Item ${i + 1}`;

  // We append to the fragment, not the live list.
  fragment.appendChild(newItem);
}

// 3. Now, attach the entire finished result to the live DOM in ONE single
operation.
list.appendChild(fragment); // This causes only ONE reflow and repaint!

// STOP THE TIMER
console.timeEnd("Loop WITH fragment");
```

If you run both of these code blocks, you will see that the version using the **DocumentFragment** is significantly faster..

The "Magic" of Appending a Fragment

When you append a **DocumentFragment** to another node, a special thing happens:

- The **DocumentFragment** itself is not added to the DOM tree.
- Instead, all of the children of the fragment are moved into the target element.
- The DocumentFragment is left empty.

This makes it a perfect, single-use container for batching DOM updates.

Summary

Feature	DocumentFragment
What is it?	An in-memory, "offline" DOM container.
Why use it?	To batch multiple DOM updates (adds, moves) into a single operation.
The Benefit	Dramatically improves performance by minimizing the number of expensive reflows and repaints .
How to use it?	1. document.createDocumentFragment(). 2. Append all your new elements to the fragment in a loop. 3. Append the single fragment to the live DOM element.
When should I use it?	Always use a DocumentFragment when you need to add a large number of elements to the DOM in a loop.